Scheduling and Fluid Routing for Flow-Based Microfluidic Laboratories-on-a-Chip

Minhass, Wajid Hassan; McDaniel, Jeffrey; Raagaard, Michael Lander; Brisk, Philip; Pop, Paul; Madsen, Jan

Published in:
I E E E Transactions on Computer - Aided Design of Integrated Circuits and Systems

Link to article, DOI:
10.1109/TCAD.2017.2729463

Publication date:
2017

Document Version
Peer reviewed version

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Scheduling and Fluid Routing for Flow-Based Microfluidic Laboratories-on-a-Chip

Wajid Hassan Minhass*, Jeffrey McDaniel†, Michael Raagaard*, Philip Brisk†, Paul Pop* and Jan Madsen*

***Abstract*—Microfluidic laboratories-on-chip (LoCs) are replacing the conventional biochemical analyzers and are able to integrate the necessary functions for biochemical analysis on-chip. There are several types of LoCs, each having its advantages and limitations. In this paper we are interested in flow-based LoCs, in which a continuous flow of liquid is manipulated using integrated microvalves. By combining several microvalves, more complex units, such as micropumps, switches, mixers and multiplexers, can be built. We consider that the architecture of the LoC is given, and we are interested in synthesizing an implementation, consisting of the binding of operations in the application to the functional units of the architecture, the scheduling of operations and the routing and scheduling of the fluid flows, such that the application completion time is minimized. To solve this problem, we propose a List Scheduling-based Application Mapping (LSAM) framework and evaluate it by using real-life as well as synthetic benchmarks. When biochemical applications contain fluids that may adsorb on the substrate on which they are transported, the solution is to use rinsing operations for contamination avoidance. Hence, we also propose a rinsing heuristic, which has been integrated in the LSAM framework.**

I. Introduction

LAboratories-on-a-chip (LoCs) integrate multiple biochemical processing components (e.g., dispensers, filters, mixers, separators, detectors) into a single device, shrinking benchtop-scale chemical and biological analysis to the sub-millimeter scale [1]. Compared to conventional biochemical analyzers, LoCs offer advantages such as reduced sample and reagent volume, faster and higher throughput biochemical processing, and ultra-sensitive detection, with multiple assays (biochemical "algorithms") being integrated into the same chip [2]. LoCs have been proposed for many applications, including clinical and point-of-care diagnostics, prenatal screening, automated drug discovery, DNA analysis, enzymatic and proteomic analysis, cancer and stem cell research, food control testing, environmental monitoring, and biological weapons detection, among others [1], [3]–[12].

The key technology that has enabled many LoC designs is the integrated microvalve [13]–[15]. Microvalves can be combined to form larger components such as peristaltic pumps, mixers, multiplexers, and latches, among others [1], [16], [17], while integration density has advanced faster than Moore's Law [18]. For example, a commercial LoC featuring 25,000 integrated microvalves that can run 9,216 polymerase chain reactions in parallel has been available since 2008 [19].

Historically, LoCs have been designed manually using CAD/drawing software such as AutoCAD[1] and SolidWorks[2]. LoC designers could benefit from fully automated software tools that adapt semiconductor VLSI/CAD algorithms and design methodologies to microfluidics. Initial efforts in this direction focused on device-level physical modeling of components [20], [21], yielding full-custom bottom-up design methodologies that remain mostly manual; once the chip is designed, the application is mapped onto the LoC based on some nebulous understanding of component functionality and application needs [22]; this process repeats each time the application or architecture changes.

A. Contribution

This paper focuses on the *application mapping* phase of an mVLSI design flow. At this point, the mVLSI device architecture has been defined and its physical topology has been laid out. The physical layout determines the length of the fluidic connections (channels) between the components that perform biochemical operations; lacking this information, the application mapper cannot determine a-priori the fluid transport latencies between components. Our application mapper adapts List Scheduling [23] to perform scheduling and binding and uses Dijkstra's algorithm [24] to compute routing paths. These heuristics are fast, scalable, and yield good quality results.

Fluids transported between components may leave residue behind in the flow channel path, which could contaminate the next fluid that is transferred through those channels. Suppose that we want to transport fluid f from component m_i to component m_j. First, component m_j must be rinsed if it has been used previously. Then, our application mapper will insert rinsing operations into the schedule to ensure that a contamination-free path $P_{i,j}$ from m_i to m_j is available for f. After transporting fluid from m_i to m_j, the mapper does not immediately rinse m_i; instead, the mapper waits until a new operation is bound to m_i at a later point in the schedule before rinsing m_i. To the best of our knowledge, this paper represents the first effort to integrate scheduling and rinsing into the mVLSI application mapping process.

Second, our mapper eliminates an unrealistic assumption about fluid transport that has been present in all prior work. Suppose that we want to transport fluid f from m_i to m_j.

*Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark. e-mail: paupo@dtu.dk and

†Department of Computer Science and Engineering, University of California, Riverside, Riverside, California, USA. e-mail: jmcda001@ucr.edu, philip@cs.ucr.edu.

[1]http://www.autodesk.com/products/autocad/overview

[2]http://www.solidworks.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Simply finding a routing path $p_{i,j}$ and opening all microvalves on the path is insufficient; although there will be some natural diffusion of fluid from m_i into the open channels, the actual transport of f's full quantity must be actuated by pressure. Therefore, it is also necessary to compute sub-paths $p_{In,i}$ from an input port m_{In} to m_i and $p_{j,Out}$ from m_j to an output port m_{Out}. This creates a longer path $P_{i,j} = p_{In,i} \cdot p_{i,j} \cdot p_{j,Out}$, where $\cdot$ is the path concatenation operator. m_{In} must inject a specific fluid type called *buffer*, which is a solution chosen for non-reactivity with assay fluids that also ensures pH-stability. Using buffer ensures that the transport process does not inadvertently contaminate fluid f.

Prior work on application mapping has provided each component with *implicit* input and output ports, which are not modeled in the LoC architecture. Implicit I/O ports trivialize the computation of sub-paths $p_{In,i}$ and $p_{j,Out}$, which connect m_i and m_j to their respective implicit input and and output ports. Implicit I/O ports, however, are problematic because the total number of I/O ports in an mVLSI LoC is limited as a design rule. For example, the Stanford Microfluidic Foundry[3] has a hard limit of 35 "hole punches" (flow and control I/O ports). Thus, adding implicit ports to each component is only feasible for very small LoC designs.

Our application mapper eliminates the assumption of implicit I/O ports. Each fluid transport operation uses a full flow path $P_{i,j}$, including sub-paths $p_{In,i}$ from an explicit input port m_{In} and $p_{j,Out}$ to an explicit output port m_{Out}. All rinsing operations are handled similarly: each rinse path starts at an input, includes at least one contaminated channel or component, and ends at an output. Eliminating these assumptions makes our application mapper considerably more realistic than prior work on the topic.

B. Related Work

We are primarily concerned with prior work on mVLSI application mapping. We also discuss relevant work on architecture synthesis (an automated approach to derive a system specification and schematic from a high-level specification of an assay), as well as fluid routing and rinsing. We do not discuss tangentially related problems such as control synthesis or physical design, which are beyond the scope of this work; the algorithms presented here can be integrated into an mVLSI design flow in a manner that is independent of the choice of algorithms and heuristics that solve these problems.

1) Application Mapping: This work is a direct extension of the first paper on application mapping published by Minhass et al. [25], which also employed List Scheduling. Our work differs in terms of how routing is performed: Minhass et al. explicitly enumerate different routing paths between pairs of components, which is inefficient non-scalable. In a follow-up work, Minhass et al. [26] replaced the heuristic approach with a constraint solver that simultaneously performs scheduling and resource binding, but ignoring fluid routing; this approach yields optimal solutions, but does not scale to large designs. Dinh et al. [27] introduced a similar optimal approach, based on clique finding, which includes the possibility of temporary storage using fluidic memory components.

Further optimization can be achieved through application-specific storage assignment, and temporarily "caching" fluids in channels that are temporarily unused [28]; the drawback of this approach is that channels allocated for storage cannot be used for fluid transport, possibly leading to longer routing paths and slower fluid transport times.The algorithms presented in our paper can support caching by designating each fluid channel as a storage component that can be routed through when empty.

Li et al. [29] added several new constraints to the application mapping process: immediate execution, mutual exclusion, and parallel execution. Immediate execution requires two dependent assay operations to be scheduled one-after-the-other without delay (other than fluid transport latency); mutual exclusion requires that several operations must be scheduled in distinct chip components; and parallel execution requires multiple operations to be scheduled concurrently. Although the assays that we use for evaluation do not require these constraints, it is straightforward to extend our mapping heuristic to account for them. Additionally, Li et al. describe how to modify an mVLSI application mapper to account for sieve valves, which partially close in order to trap large particles without stopping the flow of fluid and smaller particles. Particles trapped by the sieve valves may be pre-washed prior to mixing to increase the input concentration, or post-washed after mixing to collect products; pre- and post-washing of particles are *explicit* operations that are stated as part of the assay procedure. They should not be confused with the rinsing proposed in this paper for decontamination purposes. Although the assays that we use for evaluation do not require sieve valves, it is straightforward to extend our mapping heuristic to account for the transport of particles to sieve valves and to include pre- and post-washing operations as part of the routing process.

Our work differs from the preceding application mappers in two key respects. Firstly, we track contaminated components and fluid channels and include rinsing steps to decontaminate them during the fluid routing process. Second, we eliminate the assumption that each component has implicit I/O ports, thereby yielding more realistic assumptions about how fluid transport can be achieved in practice.

2) Reliability-aware Application Mapping: Tseng et al. [30], [31] developed a greedy resource binding technique that tries to minimize the number of fluid transfers. Their objective was to improve LoC reliability by minimizing the amount of valve switching required to execute the assay; in principle, this technique could also improve performance by reducing both the routing overhead and the number of rinsing steps that are needed to remove contamination.

Tseng et al. [32] introduced a reliability-aware application mapper targeting a reconfigurable 2D microvalve array. The key innovation is to reconfigure the array so that different groups of valves act as pumps at different times over the execution of a bioassay as a form of wear-leveling. To the best of our understanding, mVLSI LoCs are discarded after a single use, so we are not concerned about long-term reliability issues such as wear leveling.

[3]http://web.stanford.edu/group/foundry/Basic%20Design%20Rules.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

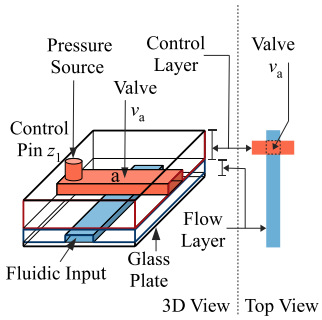


Fig. 1: Multi-layer PDMS microvalve [1], [13].

3) Architecture Synthesis: Architecture synthesis is the process of deriving an LoC architecture from a high-level assay specification. Unlike high-level synthesis for semiconductors [23], mVLSI architecture synthesis requires flow layer physical design to be performed first in order to determine fluid channel lengths, which are needed to determine routing latencies. [33], and, in our case, rinsing latencies.

Eskesen et al. [34] add redundancy during architecture synthesis so that the resulting LoC can tolerate some non-zero number of microvalve and channel failures. In principle, our mapper is compatible with any LoC architecture, regardless of whether or not redundancy is present. In principle, the presence of additional routing resources may increase the amount of fluid that can be transported in parallel; if this is possible, then our mapper can implicitly take advantage of the redundant resources when failures occur; similarly, it can map assays onto a device with some amount of failure as long as at least one routable path is available for every transport operation that is required.

4) Fluid Routing and Rinsing: Su et al. [35] developed algorithms for concurrent fluid routing in a reconfigurable microvalve array. The algorithm decomposes long routes into shorter L-shaped sub-paths. This algorithm was not integrated into a larger application mapper and seems unlikely to generalize to arbitrary planar topologies; thus it is too restrictive to be compatible with the application mapper presented here.

Hu et al. [36] developed a rinsing algorithm to decontaminate an LoC after use. To the best of our knowledge, this is the only fluid router, other than ours, which eliminates the assumption of implicit ports. One drawback is that it is requires construction of a rinsing path dictionary, similar to the prior work on application mapping by Minhass et al. [25], which suffers from scalability issues. The rinsing algorithm does account for variations in the amount of contamination in different components and channels, and different molecular species of contaminants, yielding heterogeneous rinse times; although we do not account for these effects in our implementation, it is straightforward to modify our mapper to account for them.

II. System Model

A. Technology Overview

1) Fluid Transport in PDMS Chips: A simple microfluidic chip can be formed by patterning fluid channels into one layer of polydimethylsiloxane (PDMS), a cheap, rubber-like elastomer, and mounting the PDMS layer on top of a rigid substrate, such as a glass slide. The rigid substrate provides the "floors" of each channel, while the walls and ceilings are patterned in the PDMS layer. Holes are punched into the PDMS layer to provide I/O access to the channel network.

Applying pressure from a pressurized source such as a syringe pump injects fluid into the chip. Transporting pressurized fluid through a channel displaces the fluid or gas which previously occupied the space. Thus, fluid injection can be used either to dispense a new fluid into the chip or to dispense a different fluid from one location in the chip to another. When injecting fluid, some displaced fluid at the end of the path connection must flow out of the chip; otherwise, accumulated backpressure will eventually damage the chip, rendering it unusable. Thus, every input, output, or fluid transport operation requires a connection from an input port to an output port.

2) Integrated Microvalve Technology: Microvalves are formed by stacking two or three PDMS layers on top of one another. One layer is patterned with fluid flow channels (as in the preceding subsection), while the others are patterned with control channels. As shown in Fig. 1, a microvalve is formed at points where a control channel (red) crosses a fluid channel (blue); push-down (up) valves are formed where the control channel crosses above (below) the flow channel.

The control layer is connected to an external pressure source through a control pin z_1. The microvalve is normally open ('0'; no pressure); pressurizing the control channel closes the microvalve ('1'), inducing the elastic control layer to pinch the underlying flow layer (point *a*). Thus, opening and closing valves dynamically reconfigures fluidic pathways on the mVLSI flow layer. Larger and more useful components can be formed by considering one or more microvalves in conjunction with the fluidic channels that they control [37].

The integrated microvalve is the basic building block of mVLSI technology, akin to the transistor. Although many microvalve technologies have been proposed over the past 20 years, we limit our discussion here to mVLSI valves formed using multilayer soft lithography, assuming two-layer fabrication with push-down valves [1], [13].

3) Switches: Fig. 2a depicts a single microvalve switch, which restricts/allows fluid flow in a channel. Fig. 2b and c show three- and four-microvalve switches that control fluid flow through fluid channel junctions. mVLSI LoCs can be viewed as a netlist of components which are connected by

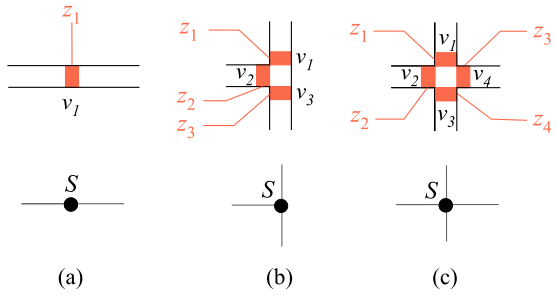


Fig. 2: Switch Configurations

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

a network of routing channels. The switches open and close fluidic pathways through the network, enabling transport of fluid from one component to another.

4) Microfluidic Mixer: mVLSI components can be constructed using channels, microvalves, and switches as building blocks. We introduce the pneumatic mixer [38] as an example. As shown in Fig. 3a, a mixer can be implemented using nine microvalves, labeled v_1 to v_9; three other valves, v_{10} to v_{12} provide an external switch to direct the flow of fluid leaving the mixer. A mixer can be viewed as a netlist of four components: the switches, s_1 to s_3 and one peristaltic pump (itself, a component comprising three microvalves connected in series), as shown in Fig. 3b. Switches s_1 and s_2 control I/O access, while s_3 directs the output of the mixer to waste or to other components in the chip (not shown). The mixer has five operational phases, as shown in Fig. 3a and c.

1) **Input 1 (Ip1):** Switches s_1 and s_2 provide an open path through the top of the mixer and s_3 opens the path to waste. Fluid f_1 flows into the top of the mixer; the excess is transported to waste.
2) **Input 2 (Ip2):** Switches s_1 and s_2 provide an open path through the bottom of the mixer (isolating the fluid on top) and s_3 opens the path to waste. Fluid f_2 flows into the bottom of the mixer; the excess is transported to waste.
3) **Mix:** Switches s_1 and s_2 close I/O access to the mixer and create a closed loop through the pump. Actuating microvalves $\{v_4, v_5, v_6\}$ in a peristaltic sequence induces pumping action, which actively mixes the two fluids in the closed loop.
4) **Output 1 (Op1):** Switches s_1 and s_2 provide an open path through the top of the mixer and s_3 opens the path into the remainder of the chip. The top half of the mixed fluid is transported into the chip for further processing.
5) **Output 2 (Op2):** Switches s_1 and s_2 provide an open path through the top of the mixer; s_3 opens the path to waste. The bottom half of the mixed fluid is transported to waste and is discarded; alternatively, it could be transported elsewhere in the chip for further processing.

The fluid samples that are to be mixed do not need to occupy the full channel length from the *Input* to the upper half of the mixer. Rather each sample may occupy a certain length on the flow channel. The process of measuring the length of each fluid sample is called *metering* and is carried out by transporting the sample between two valves that are a fixed length apart [39]. Once the top half is filled, v_7 and v_2 close, stopping the flow and blocking the fluid sample in the upper half of the mixer. Since we know the flow rate (mm/s) and the sample volume (in mm, measured in terms of length through *metering*), the time until the mixer gets filled can be easily calculated, hence optical feedback to ensure correct metering is not necessary.

B. Component Model

The microfluidic mixer, described previously, is just one of many possible components in an mVLSI LoC. We characterize each component using a dual-layer modeling framework consisting of a *flow layer model* and a *control layer model*.

TABLE I: Component Library ($\mathcal{L}$): Flow Layer Model

Component	Phases ($\mathcal{P}$)	Exec. Time (C)	H
Mixer	Ip1/ Ip2/ **Mix**/ Op1/ Op2	0.5 s	30×30
Filter	Ip/ **Filter**/ Op1/ Op2	20 s	120×30
Detector	Ip/ **Detect**/ Op	5 s	20×20
Separator	Ip1/ Ip2/ **Separate**/ Op1/ Op2	140 s	70×20
Heater	Ip/ **Heat**/ Op	20°C/s	40×15
Metering	Ip/ **Metering**/ Op1/ Op2	-	30×15
Multiplexer	Ip or Op	-	30×10
Storage	Ip or Op	-	90×30

TABLE II: Mixer: Control Layer Model

Phase	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
1. Ip1	0	0	1	0	0	0	0	0	1
2. Ip2	0	1	0	0	0	0	1	0	0
3. Mix	1	0	0	Mix	Mix	Mix	0	1	0
4. Op1	0	0	1	0	0	0	0	0	1
5. Op2	0	1	0	0	0	0	1	0	0

1) Flow Layer Model: The flow layer model $\mathcal{L} = (\mathcal{P}, C, H)$ of each component is characterized by a set of operational phases $\mathcal{P}$, execution time C, and geometrical dimensions H. Table I shows flow layer models for eight commonly utilized microfluidic components [40]. The geometrical dimensions H are given as length×width and are scaled, with a unit length being equal to 150 μm (e.g., length 10 corresponds to 1500 μm). Table I lists the different operational phases of each component: for some components, the phases must be serialized, as in the case of the mixer.

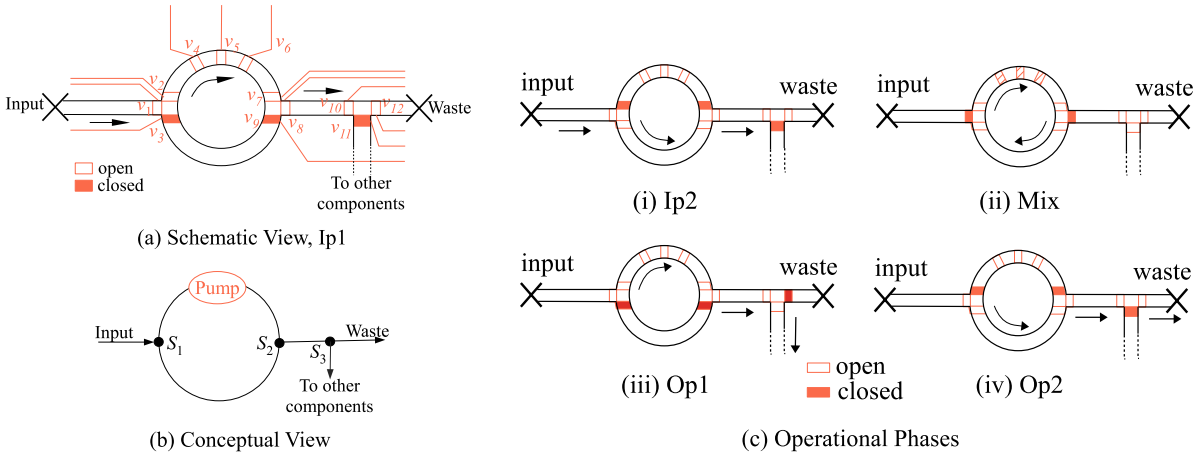
2) Control Layer Model: While the flow layer model captures the high-level behavior of each component, the the control layer model captures microvalve actuation details required to operate it. As an example, Table II presents the control layer model of the mixer shown in Fig. 3, showing the status of each microvalve during each operational state. During the "Mix" state where the peristaltic actuation sequence for valve set $\{v_4, v_5, v_6\}$ is dynamic [38].

An mVLSI LoC is typically controlled by a host PC that issues control signals at the granularity of the control layer model; the host may also perform data acquisition and signal processing operations [41] as needed.

C. Architecture Model

An mVLSI LoC architecture is modeled as a topology graph (or netlist) $\mathcal{A} = (\mathcal{N}, \mathcal{D})$, e.g., as shown in Fig. 4a, where $\mathcal{N} = \mathcal{M} \cup \mathcal{S}$ is a set of vertices ($\mathcal{M}$ is the set of components; $\mathcal{S}$ is the set of switches) and $\mathcal{D}$ is a finite set of edges representing fluid channel segments. In principle, a channel segment can support transport of fluid in either direction; however, there are cases where the direction of flow may affect the correctness of the bioassay due to imperfections in the route. Therefore, we represent each directional channel segment with a directed edge $d_{i,j} \in \mathcal{D}$ and each bidirectional channel segment with a pair of directed edges $d_{i,j}$ and $d_{j,i}$. Bidirectional fluid channel segments must satisfy two constraints: (1) fluid cannot simultaneously flow through $d_{i,j}$ and $d_{j,i}$; and (2) without loss of generality, any fluid that contaminates (rinses) $d_{i,j}$ implicitly contaminates (rinses) $d_{j,i}$ as well.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



(a) Schematic View, Ip1

(b) Conceptual View

(c) Operational Phases

Fig. 3: Microfluidic Mixer

A flow path, $P_{i,j}$, is a subset of one or more directed edges of $\mathcal{D}$, representing a directed channel between any two vertices $n_i, n_j \in \mathcal{N}$ using a chain of directed edges in $\mathcal{D}$ (e.g., in Fig. 4a, $P_{In_1,Mixer_1} = (d_{In_1,S_1}, d_{S_1,S_2}, d_{S_2,Mixer_1})$ represents a flow path from vertex In_1 to vertex $Mixer_1$). Fluid transport using a flow path is analogous to a circuit-switched network: the entire flow path is reserved until the completion of the fluid transfer (e.g., until the fluid reaches $Mixer_1$); unlike a circuit switched network, the fluid transport leaves residue, so the flow path must be rinsed before a subset of its edges can be used again for fluid transport.

Two flow paths overlap if they share at least one directed edge, $d_{i,j}$. This means that they can only be utilized in a serialized fashion (including the overhead of rinsing). For example, flow paths $P_{In_1,Mixer_1}$ and $P_{Heater_1,Mixer_1} = (d_{Heater_1,S_5}, d_{S_5,S_2}, d_{S_2,Mixer_1})$ overlap because they share $d_{S_2,Mixer_1}$.

Let $t_{route}(d_{i,j})$ be the routing latency of $d_{i,j}$, i.e., the time required for a fluid sample to traverse $d_{i,j}$. Let $t_{route}(P_{i,j}) = \sum_{d_{k,l} \in P_{i,j}} t_{route}(d_{k,l})$ denote the routing latency for flow path, $P_{i,j}$, which is the sum of the routing latencies of $P_{i,j}$'s constituent edges. We assume a flow rate of 10 mm/s [41].

D. Biochemical Application Model

We model a biochemical application using a sequencing graph [42] $\mathcal{G} = (\mathcal{O}, \mathcal{E})$, which is directed, acyclic and polar, i.e., there is a *source* vertex that has no predecessors and a *sink* vertex that has no successors; Fig. 4b shows an example.

Each vertex $o_i \in \mathcal{O}$ represents an operation that will be scheduled and bound onto an architecture component in $\mathcal{M}$, denoted by binding function $\mathcal{F}_{Op} : \mathcal{O} \rightarrow \mathcal{M}$. The latency of o_i when bound to m_j is a known constant.

The edge set, $\mathcal{E}$, models the dependency constraints in the assay, i.e., an edge $e_{i,j} \in \mathcal{E}$ from o_i to o_j indicates that the fluid output of o_i is then input to o_j. If the fluid output from o_i cannot be used immediately by o_j (e.g., it has to wait for another operation to finish on component $\mathcal{F}_{Op}(o_j)$), it has to be stored in a "storage unit", see Table I [39] or in an otherwise unused routing channel on-chip [28]. An operation has one incoming edge for each input phase of the corresponding component, and at most one edge for each output phase; if it has fewer outgoing edges than the number of output phases, the remaining fluids are transported to waste.

All inputs to a given operation must arrive at its component before an operation can execute. All outgoing edges from the *source* vertex in the sequencing graph must be bound to input reservoirs (to ensure that the correct fluids are input to the device) and all incoming edges to the *sink* vertex must be bound to output or waste reservoirs. We assume that each input operation dispenses the correct volume of fluid through metering [13]. The sequencing graph model does not explicitly represent rinsing operations for cross-contamination removal; they are inserted as part of the application mapping process.

III. Problem Formulation

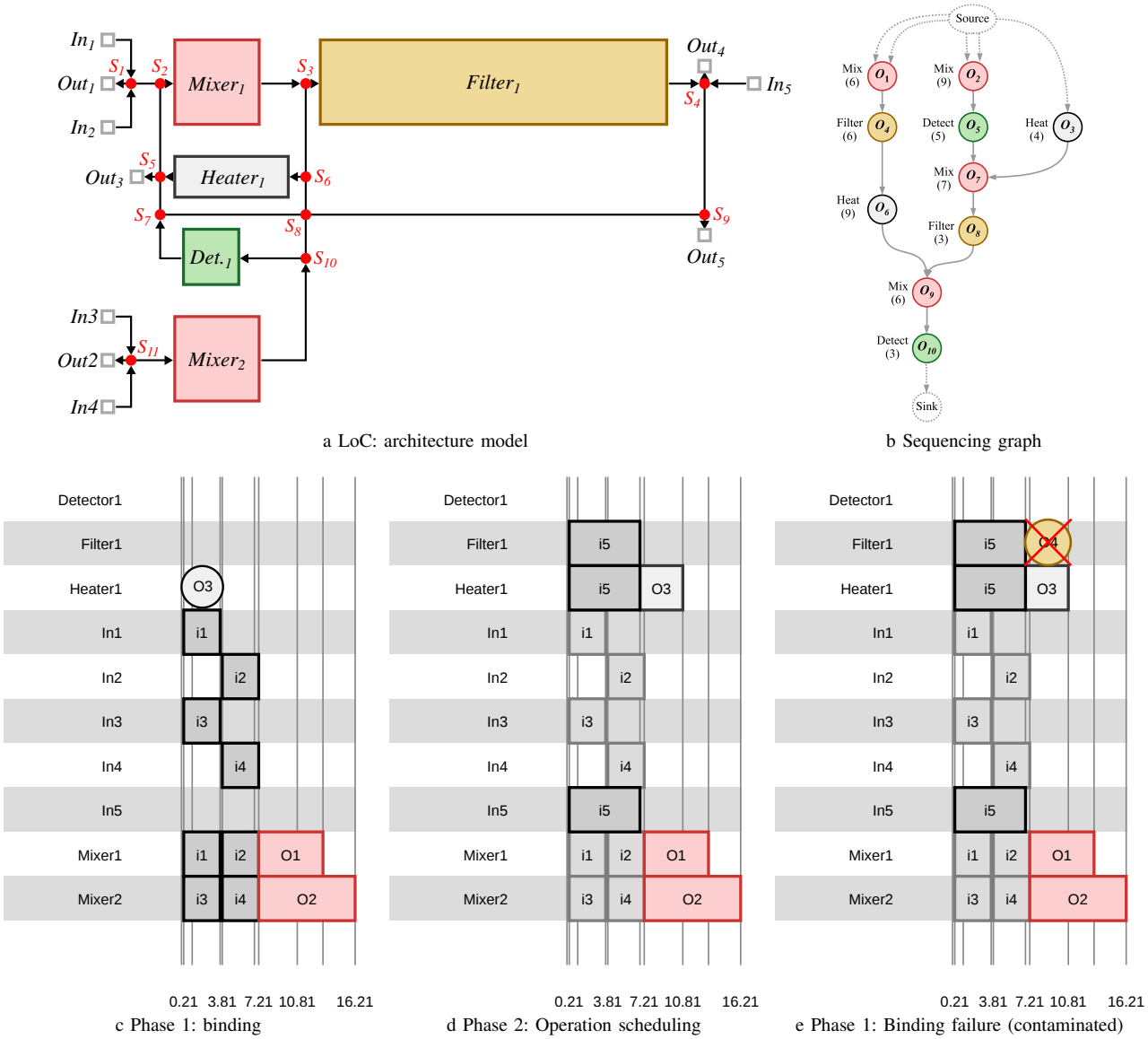
Given a characterized mVLSI component library, $\mathcal{L}$, an mVLSI LoC architecture, $\mathcal{A}$, and a bioassay modeled as a sequencing graph, $\mathcal{G}$, we wish to synthesize an implementation $\Psi = < \mathcal{T}, \mathcal{F} >$, where $\mathcal{T}$ is the schedule and $\mathcal{F}$ is the binding. Specifically, $\mathcal{T} = < \mathcal{T}_{Op}, \mathcal{T}_{Path}, \mathcal{T}_{Rinse} >$, where $\mathcal{T}_{Op}$ is the operation schedule (set of operation start times), $\mathcal{T}_{Path}$ is the schedule of fluid transfers, and $\mathcal{T}_{Rinse}$ is the schedule of rinse paths. Similarly, $\mathcal{F} = < \mathcal{F}_{Op}, \mathcal{F}_{Path}, \mathcal{F}_{Rinse} >$, where $\mathcal{F}_{Op}$ is the binding of operations to components and $\mathcal{F}_{Path}$ and $\mathcal{F}_{Rinse}$ are the respective bindings of fluid transfers and rinse paths to routing channels. The objective of the scheduler is to minimize the bioassay completion time, denoted $\delta_{\mathcal{G}}$,

Each operation, o_i has a corresponding *ready time*, $t_{ready}(o_i)$, *start time*, $t_{start}(o_i)$, and *finish time*, $t_{finish}(o_i)$. The ready time is the earliest time at which o_i may execute based on the the finishing times of o_i's predecessors; o_i's start and finish times are computed by the schedule; it follows that

TABLE III: Allocated Components ($\mathcal{M}$)

Function	Units	Notations
Input port	5	In_1, In_2, In_3, In_4, In_5
Output port	5	Out_1, Out_2, Out_3, Out_4, Out_5
Mixer	2	$Mixer_1$, $Mixer_2$
Heater	1	$Heater_1$
Filter	1	$Filter_1$
Detector	1	$Detector_1$
Storage Reservoir	8	Res_1

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



a LoC: architecture model

b Sequencing graph

c Phase 1: binding

d Phase 2: Operation scheduling

e Phase 1: Binding failure (contaminated)

Fig. 4: (a) An example LoC architecture containing two mixers, one heater, one detector, and one filter and (b) the sequencing graph representation of the experiment to be run on the LoC. The arrows from the source to the first operations are implicit fluid input operations which will be bound to input components as shown in (c-e). (c) The first two operations, Mix o_1 and Mix o_2, have been previously bound and scheduled on Mixer1 and Mixer2 respectively. The Heat o_3 is bound to Heater1 during Phase 1. (d) The fluids are then routed from their respective inputs to Heater1, the component executing o_3. The operation is then scheduled to execute once all fluids have arrived. (e) Filter operation o_4 is now ready to schedule, but Filter1 is contaminated and so it o_4 is unable execute without nullifying the results of the experiment

$t_{ready}(o_i) \leq t_{start}(o_i) < t_{finish}(o_i)$. Each dependency edge $e_{i,j}$ may be associated with a fluid transfer operation. If $\mathcal{F}_{Op}(o_i) \neq \mathcal{F}_{Op}(o_j)$, then o_i and o_j are bound to different components, and a fluid transport operation is required to move the fluid from o_i to o_j, whose respective start and finish times are $t_{start}(e_{i,j})$ and $t_{finish}(e_{i,j})$; fluid transport operations do not have ready times. If $\mathcal{F}_{Op}(o_i) = \mathcal{F}_{Op}(o_j)$, then o_i and o_j are bound to the same component, eliminating the need to transport fluid.

Rinse paths are implicit, i.e., they are not represented explicitly by operations in the sequencing graph. When a rinse path R_k is scheduled and bound, its start and finish times are denoted $t_{start}(R_k)$ and $t_{finish}(R_k)$ respectively.

A legal schedule must satisfy the following constraints:

Operational: Each operation o_i is bound to one component $m_j = \mathcal{F}_{Op}(o_i)$, and m_j is capable of executing o_i.

Dependence: For each dependency edge $e_{i,j}$, operation o_i must first finish, followed by the complete fluid transfer (if needed) followed by the start of opeation o_i, i.e.: $t_{finish}(o_i) \leq t_{start}(e_{i,j}) \leq t_{finish}(e_{i,j}) \leq t_{start}(o_j)$.

Resource: No two assay operations can be bound to the same component at the same time, i.e., if $\mathcal{F}_{Op}(o_i) = \mathcal{F}_{Op}(o_j)$ then either $t_{finish}(o_i) \leq t_{start}(o_j)$ or vice-versa.

Routing: No two flow paths (including rinse paths) can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

bound to the same component at the same time, i.e., if $\mathcal{F}_{Path}(e_{i,j}) \cap \mathcal{F}_{Path}(e_{k,l}) \neq \emptyset$, then $t_{finish}(e_{i,j}) < t_{start}(e_{k,l})$, or vice-versa. Similarly, no two fluid routing paths (including rinse paths) whose execution intervals overlap can be bound to the same channel segment.

Cross-contamination (Operations): No operation can be bound to a component at a time when the component is contaminated. Let o_i and o_j be operations whose fluids are incompatible, and let $m = \mathcal{F}_{Op}(o_i) = \mathcal{F}_{Op}(o_j)$; without loss of generality, assume that $t_{finish}(o_i) < t_{start}(o_j)$. Then there must exist a rinse path R_k that includes component m such that $t_{finish}(o_i) \leq t_{start}(R_k) < t_{finish}(R_k) \leq t_{start}(o_j)$.

Cross-contamination (Channel Segments): No flow path can be bound to a contaminated routing channel segment. This constraint is analaogous to the cross-contamination constraint for operations stated above.

The function $op(m_i)$ refers to the operation most recently scheduled and bound to component m_i; this can be treated as an inverse binding function in which the time interval during which the operation is scheduled is implicit.

IV. Scheduling, Routing and Rinsing Heuristic

We propose a List Scheduling-based Application Mapping (LSAM) heuristic to schedule and bind assay operations onto an mVLSI LoC. The inputs are a sequencing graph, $\mathcal{G}$, and an mVLSI LoC architecture $\mathcal{A}$. Initially, we will ignore the issues of cross-contamination and rinsing; we will discuss them later in Section IV-E.

A. List Scheduling Overview

List scheduling [23] inserts operations that are ready to be scheduled into priority queue Q, and dequeues them in priority order. Each operation is enqueued when the last of its predecessors have been scheduled; this ensures that vertices are dequeued in topological order, regardless of priority.

The *priority* of operation o_j is the maximum sum of operation latencies along any path in the sequencing graph from o_j to the *sink*. Intuitively, operations with high priorities are more likely to be on the critical path, so dispatching them earlier tends to reduce the length of the schedule.

When operation o_j is dequeued, it is first bound to a *qualified* component m' (i.e., m' must be capable of executing o_j) before o_j can be scheduled. Binding occurs prior to scheduling because fluid routing latencies are not known a-priori. If m' contains any fluid that will not be used by o_j, then that fluid must be bound to a routing path and transported elsewhere; similarly, any fluid required by o_j that is not presently in m' must be bound to a routing path and transported into m'; o_j can only be scheduled to execute on m' once both of the aforementioned routing latencies are determined. Once o_j is bound and scheduled, each successor is examined and inserted into Q if it becomes ready.

B. LSAM: List Scheduling-based Application Mapping

Algorithm 1 presents pseudo-code for LSAM (without support for rinsing). LSAM's main loop dequeues operation o_i, from Q and processes it in three phases:

Algorithm 1 List Scheduling-based Application Mapping (LSAM): Simplified Version with Rinsing Disabled

```
1:  function SCHEDULE(G, A)
2:      Q ← source
3:      repeat
4:          o_j ← DEQUEUE(Q)
5:          m' ← ∅, t' ← ∞
6:          ▷ Phase 1: Find best component
7:          for each qualified component, m ∈ M do
8:              t ← ∞
9:              if ∃e_{i,j} ∈ E | o_i = op(m) then
10:                 t ← t_finish(op(m))
11:             else if op(m) ≠ ∅ then
12:                 t ← t_finish(op(m)) +
                        ROUTINGESTIMATE(op(m), m, t_ready(o_j))
13:             else
14:                 t ← t_ready(o_j)
15:             t ← ROUTINGESTIMATE(o_j, m, t)
16:             if t < t' then
17:                 t' ← t, m' ← m
18:         F_Op(o_j) ← m'
19:         if op(m') ≠ ∅ then
20:             BINDTOSTORAGE(m')
21:         F_Path(o_j) ← P_{i,j} ← BINDINPUTS(o_j, m', t')
22:         ▷ Phase 2: Schedule o_j and flow paths
23:         t'' ← t'
24:         for each e_{i,j} ∈ E do
25:             T_Path(e_{i,j}) ← SCHEDULEEDGE(e_{i,j}, P_{i,j}, t')
26:             if t_finish(e_{i,j}) > t'' then
27:                 t'' ← t_finish(e_{i,j})
28:         t_start(o_j) ← t''
29:         T_Op(o_j) ← SCHEDULEOPERATION(o_j, m', t_start(o_j))
30:         ▷ Phase 3: Add ready successor to queue
31:         o_k ← SUCCESSOR(o_j)
32:         if o_k is ready then
33:             t_ready(o_k) ← max(t_finish(PREDECESSORS(o_k)))
34:             ENQUEUE(o_k, Q)
35:     until Q = ∅
36:     T ←< T_Op, T_Path >
37:     F ←< F_Op, F_Path >
38:     Ψ ←< T, F >
```

Phase 1 (lines 6–21) iterates through all qualified components $m \in \mathcal{M}$ and computes the earliest time t at which m becomes available to execute the next operation; this process is described in further detail in Section IV-C. o_j is bound to m' (line 18) whose availabe time t' is minimum among all components. In Fig. 4c, o_3 is bound to component Heater1. Phase 1 also binds fluid transport operations to routing paths, including: (1) the removal of fluid from m' that is not used by o_j (lines 19-20), and (2) transport of fluids used by o_j from elsewhere on the chip to m' (discussed in further detail in Section IV-D).

Phase 2 (lines 22–29) computes the latest time that all input

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

fluids to o_j arrive at component m', denoted t''. Clearly, $t'' \geq t'$, since fluids cannot arrive at m' before m' is ready to accept them. Phase 2 first schedules each of o_j's fluidic dependencies $e_{i,j}$; $e_{i,j}$ was bound to flow path $P_{i,j}$ during a prior iteration which scheduled and bound o_i. Line 25 schedules the fluid transport operation, yielding start and finish times $t_{start}(e_{i,j})$ and $t_{finish}(e_{i,j})$. Lines 26 and 27 then update t'' if $t_{finish}(e_{i,j}) > t''$. After scheduling each requisite transfer of fluid to m', Phase 2 sets $t_{start}(o_j)$ to t'' (line 28) and schedules o_j to execute on m' starting at $t_{start}(o_j)$ (line 29). In Fig. 4d the fluid is routed from component In5 through Filter1 to Heater1. Once it has arrived (7.21s), Heat o_3 is scheduled on Heater1.

Phase 3 (lines 30–34) enqueues successor o_k of o_j in Q if o_k is ready to be scheduled; o_k's ready time $t_{ready}(o_k)$ is initialized to the latest finish time among its predecessors (line 33). In Fig. 4e, Filter o_4 is ready at time 0, but the filter is occupied until time 7.21s, at which point it is contaminated, thereby stalling the execution of the assay.

Rinsing complicates scheduling and binding. Contaminated qualified components are unavailable for binding during Phase 1. Contaminated channels and components that support route-through are likewise unavailable, which makes it more difficult for Phase 1 to accurately estimate routing overhead (lines 12 and 15). Section IV-E describes the necessary enhancements to LSAM to integrate rinsing support.

C. Operation Binding: Details

During Phase 1, there are three cases to consider when determining the time t at which qualified component m could become available to execute operation o_j:

Case 1 (lines 9–10): $op(m)$ produces one output fluid that is input to o_j: o_j must wait for operation $op(m)$ to finish (line 10). The requisite fluid is present, so m is ready.

Case 2 (lines 11–12): $op(m)$ generates at least one fluid that is *not* input to o_j. This fluid must be transferred to storage, freeing m to execute o_j; the fluid transfer latency is not known until a future iteration that will bind the storage operation to a qualified storage component. This binding decision will *only* be made if o_j is bound to m. Instead of making a future binding decision a-priori, function ROUTINGESTIMATE estimates the routing latency (line 12). In this case, t depends on $t_{finish}(op(m))$ plus the estimated routing overhead (line 12).

Case 3 (lines 13–14): m is not performing any operation and contains no fluid; t is estimated to be $t_{ready}(o_j)$, as the component does not constrain the schedule.

After these three cases the estimate of routing fluids used by o_j to m is added to t (line 15). After selecting the qualified component m' that minimizes t, Phase 1 wraps up by establishing the necessary bindings to enable Phase 2 to precisely determine $t_{start}(o_j)$, and then schedule o_j.

If m' was selected via Case 2, then the decision to transfer fluid from m' to storage becomes permanent (lines 19-20); however, LSAM delays storage binding until a future iteration. To simplify notation, let $o_x = op(m')$. Then for each successor o_y of o_x (excluding o_j), the function BINDTOSTORAGE (line 20): (1) removes dependency edge $e_{x,y}$ from $\mathcal{G}$; (2) inserts an explicit storage operation, o_{store}, into $\mathcal{G}$; and (3) inserts new dependency edges $e_{x,store}$ and $e_{store,y}$ into $\mathcal{G}$. At the completion of Phase 1, Component m' becomes the target for each fluidic dependency edge $e_{i,j}$. The function BINDINPUTS computes a flow path $P_{i,j}$ and binds $e_{i,j}$ to $P_{i,j}$ (line 21).

D. Flow Path Scheduling and Binding

Consider fluidic dependency $e_{i,j} \in \mathcal{E}$, and let $\mathcal{F}_{Op}(o_i) = m_i$ and $\mathcal{F}_{Op}(o_j) = m_j$; $t_{finish}(o_i)$ is known from a prior LSAM iteration; $t_{start}(o_j)$ cannot be known until the latencies of the fluid transport operations that deliver its inputs are determined.

The first step is to bind $e_{i,j}$ to a flow path. Simply finding a path in the architecture from component m_i to m_j is insufficient: it is necessary to select an input port m_{In} to provide buffer fluid along with an output port m_{Out}, and concatenate to three sub-paths: $P_{i,j} = p_{In,i} \cdot p_{i,j} \cdot p_{j,Out}$.

We use Dijkstra's Algorithm [24] to compute the desired flow path. First, we augment the architecture graph with a super-source connected to each buffer input and a super-sink connected to each output. We invoke Dijkstra's Algorithm three times to compute the three sub-paths: (1) from the super source to m_i, which implicitly selects an input port m_{In} (dropping the super-source from this sub-path yields $p_{In,i}$); (2) sub-path $p_{i,j}$ from m_i to m_j; and (3) from m_i to the super-sink, which implicitly selects the output port m_{Out} (dropping the super-sink from this sub-path yields $p_{j,Out}$). The three flow paths are then concatenated to form $P_{i,j}$.

Dijkstra's Algorithm is restricted to avoid components or channels that are presently contaminated or are otherwise in use. It is allowed to route through components, noting that certain components should be avoided if they affect bioassay functionality. For example, routing through a heater could affect the fluid's temperature, or routing through a filter could affect its composition; on the other hand, routing through a clean and inactive mixer would not alter the fluid.

Once the flow path is computed, the next step is to determine the start and end times of the fluid transfer, $t_{start}(e_{i,j})$ and $t_{finish}(e_{i,j})$. $t_{start}(e_{i,j})$ is the first time that the following three conditions are met: (1) m_i is no longer executing an operation. (2) m_j is not executing an operation and has been drained of fluids. (3) All channel segments $d_{k,l} \in P_{i,j}$ are available and are uncontaminated. This point in time is calculated as:

$$t_{start}(e_{i,j}) = max(t_{finish}(o_i), t_{ready}(o_j), \{t_{finish}(d_{k,l}) | d_{k,l} \in P_{i,j}\}),$$

where $t_{finish}(d_{k,l})$ is the finish time for any previously scheduled and bound fluid transport operation, other than $e_{i,j}$, that uses channel segment $d_{k,l}$.

The finish time of the flow is then given by:

$$t_{finish}(e_{i,j}) = t_{start}(e_{i,j}) + t_{route}(P_{i,j})$$

where $t_{route}(P_{i,j})$ is the routing latency using flow path $P_{i,j}$.

A flow path is computed via three invocations of Dijkstra's Algorithm per dependency edge $e_{i,j}$. Since flow paths must be mutually exclusive, once a path $P_{i,j}$ is found, each channel $d_{k,l} \in P_{i,j}$ is marked as busy during the time interval

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

$[t_{start}(e_{i,j}), t_{finish}(e_{i,j})]$ when the fluid transfer is scheduled; after that, $d_{k,l}$ is marked as contaminated until it is rinsed.

The subroutine SCHEDULEEDGE (line 25) binds the dependency edge $e_{i,j}$ to flow path $P_{i,j}$, and schedules its start and finish times $t_{start}(e_{i,j})$ and $t_{finish}(e_{i,j})$. At the finish time, no fluids remain in the source component m_i. We set $op(m_i)$ to $\emptyset$, which indicates that no operation is currently occupying it. In principle, it is not yet safe to schedule another operation onto m_i until it has been rinsed.

E. Rinsing Heuristic

When properly modeling contamination, LSAM will eventually fail as contaminated components and channels accumulate: either all qualified components (i.e., the for-loop spanning lines 7-17) will be contaminated or Dijkstra's Algorithm will be unable to find a valid flow path for some dependency edge $e_{i,j}$ (line 25; discussed in the preceding subsection). To rectify this situation, this section presents two techniques for decontamination: *Naïve Rinsing (NR)*, and a more sophisticated *Rinsing Integrated Scheduling (RINS)*. Both NR and RINS share a common rinse path computation subroutine, which is discussed next; they differ in terms of the conditions that trigger the introduction of rinse steps into the schedule and whether or not bioassay execution pauses during rinsing.

Rinses are *implicit* operations in the sense that vertices that represent them are *never* inserted into the sequencing graph $\mathcal{G}$, although a rinse path is explicitly scheduled ($\mathcal{T}_{Rinse}$) and bound to an mVLSI channel ($\mathcal{F}_{Rinse}$). Formally, each rinse operation is a tuple $R_i = < P_i^{Rinse}, t_{start}(P_i^{Rinse}), t_{finish}(P_i^{Rinse}) >$, where P_i^{Rinse} is the rinsing path with start and finish times $t_{start}(P_i^{Rinse})$ and $t_{finish}(P_i^{Rinse})$; index i corresponds to the rinse operation and *not* to a sequencing graph operation o_i.

1) Rinse Path Computation: Let $\mathcal{C}_{\mathcal{M}}$ be the set of contaminated components and $\mathcal{C}_{\mathcal{D}}$ be the set of contaminated routing channels. When rinsing is triggered, both NR and RINSrinse as many contaminated components and routing channels as possible, regardless of whether they will be used later; this maximizes the leeway provied to the scheduler. Rinsing flushes any "dead fluid" that remains in the chip as well as its residue.

The first step is to rinse components. For each component $m_j \in \mathcal{C}_{\mathcal{M}}$, LSAM invokes Dijkstra's Algorithm to compute a flow path $P_i^{Rinse} = p_{In,j} \cdot p_{j,Out}$ from a buffer input m_{In} to an output port m_{Out}, with the restriction that P_i^{Rinse} cannot go through any component that actively holds fluid. Any other contaminated components or fluid routing channels on P_i^{Rinse} are respectively removed from $\mathcal{C}_{\mathcal{M}}$ and $\mathcal{C}_{\mathcal{D}}$, and P_i^{Rinse} is bound, i.e., $\mathcal{F}_{Rinse}(R_i) \leftarrow P_i^{Rinse}$; if no such rinsing path can be found (e.g., because all paths that include m_j also include a component that actively holds fluid), then m_j will remain contaminated until a future rinsing step.

The process then repeats for each contaminated fluid routing channel $d_{k,l} \in \mathcal{C}_{\mathcal{D}}$; for similar reasons, it may not be possible to rinse every contaminated edge in $\mathcal{C}_{\mathcal{D}}$.

2) Naïve Rinsing (NR): A bioassay operation o_j may fail to bind to an otherwise-qualified component m because (1) m is contaminated; or (2) path binding fails for at least one of its inputs (if bound to m) due to contaminated routing channels. LSAM triggers the rinse path computation procedure described in the prceding subsection if all assay operations presently in Q fail to bind. All presently-executing operations run to completion; bioassay execution pauses during rinsing.

Next, the computed rinse paths are scheduled to execute concurrently. Incompatible rinse paths, such as those that route fluid through the same channel, but in opposite direction, are sequentialized. Once rinsing completes, bioassay execution resumes. LSAM delays the start times of all not-yet-scheduled operations at least the finishing time of the current rinse step. The latency of rinsing is constrained by the latest finishing time among all of the scheduled rinse paths. This provides the rinse path schedule, $\mathcal{T}_{Rinse}$.

Naive Rinsing is simple, but does does not allow concurrent scheduling of bioassay operations and rinsing steps. For example, Fig. 5a shows the schedule obtained by LSAM+NR for the sequencing graph in Fig. 4b and the mVLSI netlist in Fig. 4a. The blue rectangles labeled with R_i represent rinsing.

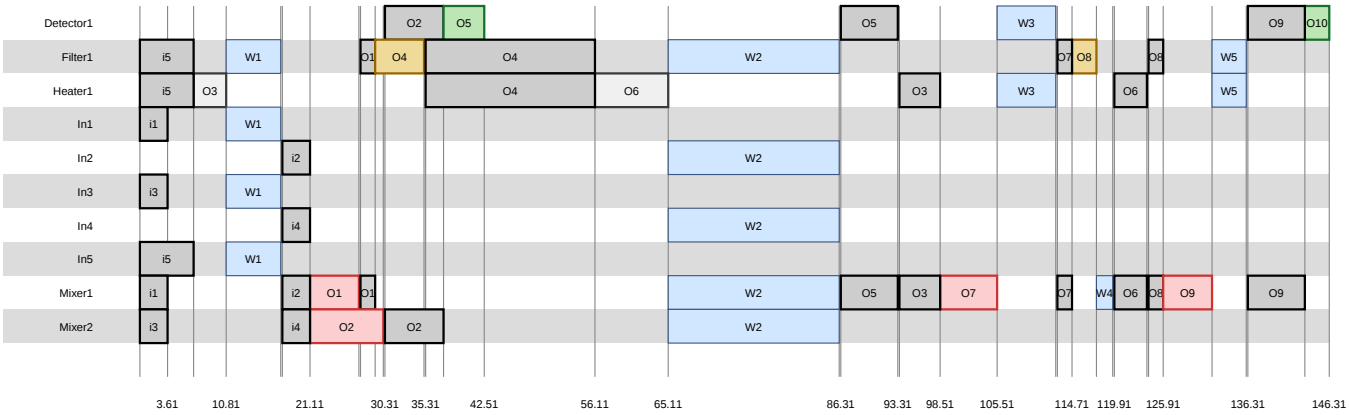
3) Rinsing Integrated Scheduling (RINS): RINS allows rinse paths to be scheduled concurrently with ongoing assay operations, and triggers rinsing operations on path binding failures, not component binding failures. Under RINS, LSAM may bind assay operations to qualified components that *are* contaminated, because all incoming paths are guaranteed to fail during binding. A path binding failure does *not* cause component binding to fail, and does not delay the schedule of bound paths that transport fluid to the target component.

For example, suppose that assay operation o_j is bound to component m'. Suppose that o_j has two predecessors, o_{i_1} and o_{i_2}. Without loss of generality, suppose that flow path $P_{i_1,j}$ binds successfully, but flow path $P_{i_2,j}$ does not. $P_{i_1,j}$ is scheduled immediately, but $P_{i_2,j}$ and operation o_j must wait until the chip is rinsed. LSAM will schedule rinsing oncurrently with $P_{i_1,j}$, or afterward.

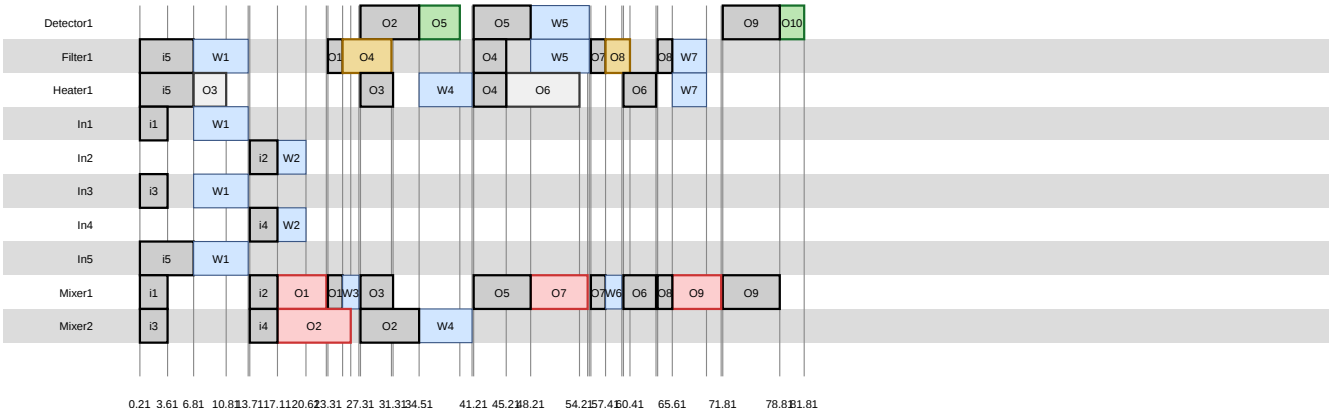
Rinse paths are computed as described in Section IV-E1, and are scheduled as early as possible (as per the criteria outlined in Section IV-D). Opportunistically, this allows rinse paths to be scheduled concurrently with bioassay operations that were scheduled previously. Once the rinse paths are scheduled, LSAM continues as described above. Compared to NR, RINS tends to have more frequent and shorter rinse steps, and, although some bioassay operations may be delayed due to rinsing, progress of the entire bioassay is rarely stalled across the entire chip.

Fig. 5b shows the schedule that is produced by LSAM when using RINS for rinsing compared to NR in Fig. 5a. RINS reduces the total assay execution time from 146.31 s to 81.81 s. As an example of the difference, in Fig. 5a NR schedules o_3 before a rinse step; in Fig. 5a, o_3 executes concurrently with the rinsing. As the assay proceeds, RINS saves more and more time by interleaving operations with rinse steps; meanwhile, the delays imposed by rinsing under NR add more and more latency to the schedule. The result is that RINS reduces total rinse time by 44% compared to the schedule produced by NR.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



a Rinsing using the naïve rinsing (NR) method



b Rinsing using the combined scheduling and routing (RINS) method

Fig. 5: Simultaneous scheduling and rinsing for the application in Fig. 4b mapped on the architecture in Fig. 4a

V. Experimental Evaluation

A. Experimental Setup and Benchmarks

We evaluate LSAM by synthesizing two real-life bioassays, five larger synthetic benchmarks[4], and the example application (EA) from Fig. 4b (also synthetic) onto different LoC architectures. The first real-life bioassay is *multiplexed in-vitro diagnostics (IVD)*, which performs 12 operations and can is used to test different fluid samples from the human body. Each mixing operation has an execution time of 4 s and the detection operation 7 s. The second real-life bioassay is a mixing tree phase of the *polymerase chain reaction (PCR)*, used for DNA amplification. The PCR mixing tree has 7 mixing operations, each of which has an execution time of 4 s.

Each benchmark is synthesized onto one, two or three different LoC architectures with a varying number of components; if the scheduler is effective, bioassay execution times will be reduced when more components are allocated.

The algorithms were implemented in C++, and were executed on a Lenovo T400s ThinkPad with an Intel Core 2 Duo Processor running at 2.53 GHz and 4 GB of RAM.

[4]https://sites.google.com/site/biochipsimulator/Files

TABLE IV: Architectures and execution time for the real-life benchmarks scheduled by LSAM

Architecture	Ops	Input Ports	Output Ports	Allocated Components	δ_{G-LSAM}
PCR-1	7	2	2	(2,0,0,0)	39.81s
PCR-2	7	3	3	(3,0,0,0)	39.41s
PCR-3	7	4	4	(4,0,0,0)	34.61s
IVD-1	12	2	2	(2,0,0,2)	69.41s
IVD-2	12	6	6	(6,0,0,6)	45.61s

B. LSAM Evaluation

The initial set of experiments evaluate the feasibility of LSAM while ignoring the issues of cross-contamination and rinsing. Tables IV and V report the bioassay execution time, δ_{G-LSAM} (in seconds) that LSAM obtains for each benchmark. LSAM's runtime for all benchmarks was less than one second.

As a general trend, increasing the number of mixers, detectors, and I/O ports directly influences the bioassay execution time. For example, switching from the IVD-1 architecture to IVD-2 reduces δ_{G-LSAM} from 69.41 s to 45.61 s.

PCR is a more interesting case: the binary mixing tree has a layer of four mixing operations, followed by a layer

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE V: Architectures and execution time for the synthetic benchmarks scheduled by LSAM

Architecture	Ops	Input Ports	Output Ports	Allocated Components	δ_{G-LSAM}
Synthetic1-1	10	1	1	(4,2,2,2)	56.21s
Synthetic1-2	10	2	2	(4,2,2,2)	54.01s
Synthetic2-1	20	1	1	(12,4,3,1)	81.51s
Synthetic2-2	20	2	2	(12,4,3,1)	75.01s
Synthetic3-1	30	1	1	(17,6,4,3)	97.41s
Synthetic3-2	30	12	1	(17,6,4,3)	87.61s
Synthetic4-1	40	2	1	(21,9,6,4)	121.61s
Synthetic4-2	40	15	1	(21,9,6,4)	105.81s
Synthetic5-1	50	2	1	(26,12,7,5)	141.01s
Synthetic5-2	50	17	1	(26,12,7,5)	113.41s
EA-1	10	4	1	(3,1,1,0)	67.21s

of two, followed by one. For each mVLSI LoC architecture, the number of input and output ports is equal to the number of mixers, which ensures that bioassay execution is not I/O-constrained. With two available mixers (PCR-1), the mixing tree executes in four steps (two steps for the first layer, and one step each for the second and third layers); adding a third mixer (PCR-2) yielded marginal improvements, since four mixing steps were still required (the small reduction in bioassay execution time was due to reduced routing latencies); with four mixers (PCR-3), the number of steps was reduced from four to three, yielding a 5.2s improvement.

For Synthetic-1 and -2 (10 and 20 operations, respectively), we vary the number of input and output ports, keeping the allocation of other components constant. For both Synthetic-1 and -2, increasing the number of I/O ports yields a small, but noticeable, improvement in δ_{G-LSAM}.

For Synthetic-3, -4, and -5 (30, 40, and 50 operations), we dramatically increase the number of input ports to a sufficient number of provide all input fluids in parallel, while keeping the number of output ports and other allocated components constant. For these benchmarks, increasing the input ports reduces δ_{G-LSAM} by approximately 10%, 12%, and 19% respectively. Clearly, δ_{G-LSAM} is dominated by bioassay operation latencies, but the contribution of I/O to δ_{G-LSAM} remains non-negligible.

Varying the number of I/O ports and components directly influences the chip area and also δ_{G-LSAM}. Chip area is an important parameter for the chips that need to be placed in small chambers, e.g., under microscopes for detection. These results demonstrate how an mVLSI LoC design can quickly evaluate the performance of different architectures and make early-stage design decisions involving tradeoffs between performance and other relevant metrics such as cost.

C. Comparison with Constraint Programming

The second set of experiments compared LSAM to a *Constrained Programming-based Application Mapping (CPAM)* approach [26]. Constraint programming yields optimal solutions, but the solvers typically take a long time to converge, especially for large problem instances. As described in Ref. [26], the CPAM problem formulation assumes the existence of implicit I/O ports allocated to each component and does not incorporate routing latencies into the schedule; moreover, it does not model cross-contamination and rinsing. To enable a fair comparison, we execute LSAM using the same assumptions, and limit our evaluation to small problem instances (PCR, IVD, and EA on three LoC architectures each).

TABLE VI: Experimental Results: LSAM vs CPAM [26]

Appl.	Allocated Components	LSAM $\delta_{\mathcal{G}}$	LSAM Exec. Time	CPAM [26] $\delta_{\mathcal{G}}$	CPAM [26] Exec. Time
PCR	(2, 0, 0, 0)	16 s	$<$ 0.1 s	16 s	0.2 s
	(3, 0, 0, 0)	16 s	$<$ 0.1 s	16 s	0.2 s
	(4, 0, 0, 0)	12 s	$<$ 0.1 s	12 s	0.2 s
IVD	(2, 0, 0, 2)	25 s	$<$ 0.1 s	25 s	2.6 s
	(5, 0, 0, 5)	18 s	$<$ 0.1 s	18 s	38 min 28 s
	(6, 0, 0, 6)	11 s	$<$ 0.1 s	11 s	1 min 38 s
EA	(2, 1, 1, 0)	19 s	$<$ 0.1 s	19 s	0.3 s
	(3, 1, 1, 0)	17 s	$<$ 0.1 s	17 s	0.4 s
	(4, 2, 1, 0)	17 s	$<$ 0.1 s	17 s	0.5 s

TABLE VII: Experimental Results: LSAM vs CAM [27]

Appl.	I/O Ports	LSAM $\delta_{\mathcal{G}}$	LSAM Exec. Time	CAM [27] $\delta_{\mathcal{G}}$	CAM [27] Exec. Time
PCR	(2, 2)	30.3 s	$<$ 1 s	29.8 s	12 s
IVD	(2, 2)	31.3 s	$<$ 1 s	30.5 s	39 min 2 s
EA	(2, 2)	56 s	$<$ 1 s	51.5 s	21 min

Table VI reports the results of the experiment. In all nine cases, LSAM obtained the same optimal result as CPAM, while running significantly faster. In the most dramatic case (IVD with 5 mixers and 5 detectors), LSAM produced a solution in less than one second while CPAM took 38 minutes and 28 seconds. These results suggest that CPAM will have even more trouble scaling, both to larger problem instances and to more realistic application mapping scenarios that eschew implicit I/O ports and include cross-contamination and rinsing.

Table VI also demonstrates situations in which increasing the number of components yields negligible improvements in bioassay execution time. Taking EA as an example, increasing the number of mixers from 3 to 4 and the number of heaters from 1 to 2 does not improve $\delta_{\mathcal{G}}$; this type of information can assist an mVLSI LoC designer to avoid inefficient portions of the design space to explore.

D. Comparison with a Clique-based Approach

We also compare LSAM to a *Clique-based Application Mapper (CAM)* [27] which suffers from similar scalability problems as CPAM. The CAM accounts for fluid routing latencies in the schedule, but assumes the existence of implicit I/O ports and does not model cross-contamination or rinsing. The results for CAM are available for the benchmarks listed in Table VII[5] but only for one target mVLSI architecture. In these experiments, CAM yields shorter assay execution times than LSAM, but runs considerably longer (up to 39 minutes, 2 seconds, compared to less than 1 second for LSAM).

[5]Ref. [27] does not include experimental results; however, results were reported in the oral presentation at ASPDAC 2013. Slides are available at: http://www.aspdac.com/aspdac2013/archive/pdf/3A-1.pdf

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE VIII: NR vs RINS results: application mapping and contamination avoidance (rinsing)

	Benchmark			
	Assay	Architecture	$\delta_{\mathcal{G}-\mathrm{NR}}$	$\delta_{\mathcal{G}-RINS}$
Real-life	IVD	IVD-1	225.61 s	200.01 s
	IVD	IVD-2	130.81 s	103.21 s
	PCR	PCR-1	97.71 s	93.81 s
	PCR	PCR-2	88.41 s	76.51 s
	PCR	PCR-3	81.11 s	66.41 s
Synthetic	Synthetic1	10-1	174.10 s	97.91 s
	Synthetic1	10-2	161.31 s	101.01 s
	Synthetic2	20-1	442.91 s	326.61 s
	Synthetic2	20-2	447.81 s	257.71 s
	Synthetic3	30-1	445.61 s	397.31 s
	Synthetic3	30-2	362.61 s	262.31 s
	Synthetic4	40-1	835.31 s	559.61 s
	Synthetic4	40-2	451.11 s	415.71 s
	Synthetic5	50-1	912.41 s	700.31 s
	Synthetic5	50-2	905.91 s	784.11 s
	EA	EA-1	146.31 s	81.81 s

E. Evaluation of Rinsing Heuristics

The last experiments compares LSAM using the NR and RINS variants for rinsing. These experiments assume the existence of explicit, rather than implicit, I/O ports, account for routing latencies, and model cross-contamination and rinsing. To the best of our knowledge, no other previously-published mVLSI application mappers include all three of these features, so a comparison to prior work is infeasible. We use the same set of real-life and synthetic benchmarks and target mVLSI architectures as our initial set of experiments.

Table VIII reports the results of these experiments. Bioassay assay execution times for NR $\delta_{\mathcal{G}-\mathrm{NR}}$ are significantly greater than for RINS $\delta_{\mathcal{G}-RINS}$ in all cases. Reductions in bioassay execution time range from 3.99% (PCR-1) to 44.1% (EA), with an average of 23.4%. These results clearly indicate that RINS is superior to NR, validating the decision to interleave rinsing operations with bioassay execution, rather than temporarily pausing bioassay execution.

Although the results reported in the preceding subsections include direct comparisons to previous work, we wish to stress the observation that only LSAM+NR and LSAM+RINS model realistic scenarios; CPAM and CAM cannot successfully target architectures that do not feature implicit I/O ports and, even for those architecture, would generate scheduling and binding results that feature cross-contamination, therefore yielding incorrect biological outcomes. The same is true of other application mappers cited in Section I-B that do not adequately account for these features.

VI. Limitations

The algorithms presented in this paper have integrated realistic fluid transport constraints and rinsing into mVLSI application mapping. This section discusses the limitations of this paper's contributions and outlines several avenues for further investigation that build upon these insights.

A. Benchmarks

Although the benchmarks used in this experimental evaluation are publicly available, it is unclear if they represent the state of the art in mVLSI applications, or whether or not they represent technically challenging problem instances that could provide a fertile proving ground for further algorithmic development. Although beyond the scope of this paper, we encourage the mVLSI CAD research community to engage in further benchmark efforts along two synergistic axes:

Representative benchmarks: assay specifications and mVLSI netlists corresponding to published mVLSI chips reported in the scientific literature. This set of benchmarks would maximize the relevant of mVLSI CAD research to users in academia and industry.

Challenge benchmarks: sets of problem instances, possibly synthetic, designed to challenge state-of-the-art algorithms; ideally, they could be generated in such a way that optimal solutions are known. This would be particularly useful to benchmark the performance of efficient heuristics for large problem instances where optimal algorithms (e.g., ILP, CSP, branch-and-bound search, etc.) cannot be expected to converge in a reasonable amount of time.

B. Open Challenges in Operation Binding

LSAM binds operations to components prior to scheduling. This decision was made because it is not possible to know the precise start time of an operation until the fluid transport latencies of its fluid inputs are established. LSAM necessariliy makes binding decisions with incomplete scheduling information, and, as a greedy heuristic, does not revisit them. This has the advantage of simplifying the underlying algorithmics, but increases the likelihood of sub-optimal decision-making.

As an example, suppose that component m has completed its operation and is presently holding fluid f. LSAM, meanwhile, is searching for a component to bind to operation o, and m is a compatible component. To make an informed binding decision, LSAM must consider the latency incurred by transfering f from m to some other component m'. From the perspective of o, the best option is to select the earliest-available component, which favors minimizing the transport latency of f from m to m', and this is precisely what LSAM does; however, this does not take into account whether or not m' is the best choice to temporarily store f from the perspective of further scheduling decisions made at later time steps. Further complicating matters, it is difficult to estimate which yet-to-be-scheduled operations will be critical since fluidic transport latencies in later time steps are not yet known.

From the perspective of future operations that will consume fluid f, it is likewise difficult to greedily determine whether m' was the best choice. Here, we put forth three greedy strategies, each of which has its own advantages and limitations that will not be known for certain until scheduling decisions are made at later time steps.

1) Transport f to the closest qualified component m'.
   Advantage: Minimize transport latency.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2729463, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Disadvantage: If another operation *o*' is bound to *m*' before *f* is used, then it will become necessary to transport *f* again, incurring additional overhead.

2) Transport *f* to the qualified component *m*' most likely to use *f*.

Advantage: Reduce the likelihood of serial transfers to free up components for other operations.

Disadvantage: Potentially longer transport path; at the time *o* is bound to *m*, LSAM does not know when the next operation that consumes *f* will be scheduled, and other operations may still be scheduled on *m*' in the meantime, necessitating additional fluid transfers.

3) Transport *f* to a storage component

Advantage: Simplicity; eliminates competition for operational resources, and could be a locally optimal decision for *f* if LSAM schedules it much later.

Disadvantage: If *f* is in fact used in the near-future, a direct transfer to the component *m** that uses *f* would be more efficient; depending on the architecture, the storage module may be far away from *m*, thus incurring a higher than necessary fluid transport latency.

For all practical purposes, it seems unlikely that *any* greedy strategy will be able to make locally optimal decisions in all cases. For our purposes we are using the third strategy. On the other hand, our evaluation provides no clear indication that LSAM is making poor decisions. One potential avenue for future work is to investigate algorithms that perform a limited form of backtracking (e.g., dynamic programming) could improve results in situations where a greedy heuristic such as LSAM makes a sub-optimal decision. Future work should investigate this issue in greater detail, possibly aided by synthetic benchmarks constructed specifically to exacerbate poor decisions made by greedy heuristics.

C. Open Challenges in Rinsing

When rinsing, the CSR heuristics introduced in this paper essentially tries to rinse the minimum amount of the chip that will allow contamination-free fluid transport from component *m* to *m*'. On the other hand, Ref. [36] tries to rinse as much of the chip as possible, noting that in the context of LSAM, contaminated components and channels may be blocked due to fluid held in other components. In principle, these two approaches represent two endpoints of a larger spectrum of problem formulations that differ in precisely how much of the (reachable) contaminated portions of an mVLSI chip should be rinsed during each fluid transport operation.

In general, the preferred strategy would be to minimally rinse the chip during fluid transport operations that are on the critical path, in order to minimize the impact on total bioassay completion time. In principle, non-critical rinse operations can rinse more of the chip, up to the point where they become critical; this can be beneficial if it moves the rinsing of some contaminated components off the critical path, thereby reducing rinse time and improving performance. Although ideal, this strategy is unrealistic because the critical path in the sequencing graph is not known until after the schedule is computed A greedy heuristic like LSAM cannot effectively determine how much of the chip needs to be rinsed during each fluid transfer.

In principle, it might seem appealing to apply the rinse optimization as a post-processing pass; however, this is problematic because changing rinse latencies, in turn, would change fluid transport latencies, which changes the start and finish times of the assay operations that were scheduled and bound by LSAM; moreover, the binding decisions made by LSAM were driven, in part, by fluid transport latency estimates, which suggests that the entire sequencing graph should be re-bound and rescheduled at the same time.

Similar to the conclusion of the preceding subsection, our suggestion is that future work on scheduling that avoids greedy decision-making can be extended to consider the impact of rinsing decisions on bioassay completion time; once again, it would be beneficial if benchmarks could be constructed in a manner to exacerbate the overhead incurred by poor greedy decision-making, while having known optimal solutions.

D. Rinse-Aware Architecture Synthesis

Any mVLSI application mapper should be compatible with any mVLSI chip capable of executing an assay, regardless of whether the chip itself is designed for maximal efficiency. In a typical mVLSI chip, only one or two input ports will be assigned to rinse buffers. In principle, there is room to investigate new techniques for mVLSI architeture synthesis that allocate additional rinse buffer I/O ports to the mVLSI netlist, while adhering to foundry design rules, in order to reduce the contribution of rinsing to total bioassay completion time after scheduling. Going one step further, it may also be possible to simultaneously co-optimize mVLSI architecture synthesis with application mapping. Although a detailed investigation of these ideas goes far beyond the scope of this paper, they do provide an interesting and potentially fruitful avenue for future research.

VII. Conclusions

In this paper we have presented a system-level modeling and application mapping framework for flow-based microfluidic LoCs. We have proposed a topology graph-based model to capture the LoC architecture and use a sequencing graph to model the biochemical applications. We have also proposed a computationally efficient heuristic approach (LSAM) that performs operation binding and scheduling while also taking the fluidic routing into account. It uses the application completion time minimization as the target objective and satisfies the dependency and resource constraints. We have also introduced a rinsing method (RINS) which will ensure that the operations within the assay are not contaminated, which would invalidate the results of the assay. Real-life case studies and a set of synthetic benchmarks have been mapped on various architectures to validate our approach. The proposed framework is expected to reduce human effort, enabling designers to make early design decisions by being able to evaluate their proposed architecture, minimizing the design cycle time and also facilitating programmability and automation.

ACKNOWLEDGMENT

This work was partly supported by the Danish Agency for Science, Technology and Innovation, Grant No. 2106-08-0018 "ProCell" and by NSF award 1351115.

REFERENCES

[1] T. Thorsen, S. J. Maerki, and S. R. Quake, "Microfluidic large-scale integration," *Science*, vol. 298, no. 5593, pp. 580–584, October 2002.

[2] G. M. Whitesides, "The origins and the future of microfluidics," *Nature*, vol. 442, pp. 368–373, July 2006.

[3] C. L. Hansen, M. O. A. Sommer, and S. R. Quake, "Systematic investigation of protein phase behavior with a microfluidic formulator," *Proceedings of the National Academy of Sciences USA*, vol. 101(40), pp. 14 431–14 436, 2004.

[4] J. W. Hong, Y. Chen, W. F. Anderson, and S. R. Quake, "Molecular biology on a microfluidic chip," *Journal of Physics: Condensed Matter*, vol. 18(18), pp. 691–701, 2006.

[5] J. W. Hong, V. Studer, G. Hang, W. F. Anderson, and S. R. Quake, "A nanoliter-scale nucleic acid processor with parallel architecture," *Nature Biotechnology*, vol. 22(4), pp. 435–439, 2004.

[6] C. C. Lee, A. Elizarov, C. J. Shu, Y. S. Shin, and A. N. Dooley, "Multistep synthesis of a radiolabeled imaging probe using integrated microfluidics," *Science*, vol. 310, pp. 1793–1796, 2005.

[7] J. S. Marcus, W. F. Anderson, and S. R. Quake, "Microfluidic single-cell mrna isolation and analysis," *Analytical Chemistry*, vol. 78(9), pp. 3084–3089, 2006.

[8] S. E. et. al., "Discovery of a hepatitis c target and its pharmacological inhibitors by microfluidic affinity analysis," *Nature Biotechnology*, vol. 12, pp. 1019–1027, 2008.

[9] C. D. C. et. al., "Microfluidics-based diagnostics of infectious diseases in the developing world," *Nature Medicine*, vol. 17, pp. 1015–1019, 2011.

[10] H. C. Fan, Y. J. Blumenfeld, U. Chitkara, L. Hudgins, and S. R. Quake, "Noninvasive diagnosis of fetal aneuploidy by shotgun sequencing dna from maternal blood," *Proceedings of the National Academy of Sciences USA*, vol. 105(42), pp. 16 266–16 271, 2008.

[11] "Verinata Health," http://www.verinata.com/.

[12] C. Fang, Y. Wang, N. T. Vu, W. Lin, Y. Hsieh, L. Rubbi, M. E. Phelps, M. Mueschen, Y. Kim, A. F. Chatziioannou, H. Tseng, and T. G. Graeber, "Integrated microfluidic and imaging platform for a kinase activity radioassay to analyze minute patient cancer samples," *Cancer Research*, vol. 70, no. 21, pp. 8299–8308, November 2010.

[13] M. A. Unger, H. Chou, T. Thorsen, A. Scherer, and S. R. Quake, "Monolithic microfabricated valves and pumps by multilayer soft lithography," *Science*, vol. 288(5463), pp. 113–116, 2000.

[14] W. H. Grover, A. M. Skelley, C. N. Liu, E. T. Lagally, and R. a. Mathies, "Monolithic membrane valves and diaphragm pumps for practical large-scale integration into glass microfluidic devices," *Sensors and Actuators B: Chemical*, vol. 89, no. 3, pp. 315–323, April 2003.

[15] I. E. Araci and S. R. Quake, "Microfluidic very large scale integration (mvlsi) with integrated micromechanical valves," *Lab Chip*, vol. 12, pp. 2830–2806, 2012.

[16] W. H. Grover, R. H. C. Ivester, E. C. Jensen, and R. A. Mathies, "Development and multiplexed control of latching pneumatic valves using microfluidic logical structures," *Lab Chip*, vol. 6, pp. 623–631, 2006. [Online]. Available: http://dx.doi.org/10.1039/B518362F

[17] J. Melin and S. Quake, "Microfluidic large-scale integration: The evolution of design rules for biological automation," *Annual Reviews in Biophysics and Biomolecular Structure*, vol. 36, pp. 213–231, 2007.

[18] J. W. Hong and S. R. Quake, "Integrated nanoliter systems," *Nature Biotechnology*, vol. 21, pp. 1179–1183, 2003.

[19] J. M. Perkel, "Microfluidics - bringing new things to life science," *Science*, November 2008.

[20] I. Klammer, A. Buchenauer, H. Fassbender, R. Schlierf, G. Dura, W. Mokwa, and U. Schnakenberg, "Numerical analysis and characterization of bionic valves for microfluidic pdms-based systems," *Journal of Micromechanics and Microengineering*, vol. 17, no. 7, pp. S122–S127, 2007.

[21] J. Siegrist, M. Amasia, N. Singh, D. Banerjee, and M. Madou, "Numerical modeling and experimental validation of uniform microchamber filling in centrifugal microfluidics," *Lab Chip*, vol. 10, pp. 876–886, 2010.

[22] W. B. Thies, "Programmable microfluidics," *presented at Stanford University*, October 2007.

[23] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[24] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[25] W. H. Minhass, P. Pop, and J. Madsen, "System-level modeling and synthesis of flow-based microfluidic biochips," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2011.

[26] ——, "Synthesis of Biochemical Applications on Flow-Based Microfluidic Biochips using Constraint Programming," in *Proc. of the IEEE Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP)*, 2012, pp. 37–41.

[27] T. A. Dinh, S. Yamashita, T. Ho, and Y. Hara-Azumi, "A clique-based approach to find binding and scheduling result in flow-based microfluidic biochips," in *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, 2013, pp. 199–204. [Online]. Available: http://dx.doi.org/10.1109/ASPDAC.2013.6509596

[28] T. Tseng, B. Li, U. Schlichtmann, and T. Ho, "Storage and caching: Synthesis of flow-based microfluidic biochips," *IEEE Design & Test*, vol. 32, no. 6, pp. 69–75, 2015. [Online]. Available: http://dx.doi.org/10.1109/MDAT.2015.2492473

[29] M. Li, T. Tseng, B. Li, T. Ho, and U. Schlichtmann, "Sieve-valve-aware synthesis of flow-based microfluidic biochips considering specific biological execution limitations," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, 2016, pp. 624–629.

[30] K. Tseng, S. You, W. H. Minhass, T. Ho, and P. Pop, "A network-flow based valve-switching aware binding algorithm for flow-based microfluidic biochips," in *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, 2013, pp. 213–218. [Online]. Available: http://dx.doi.org/10.1109/ASPDAC.2013.6509598

[31] K.-H. Tseng, S.-C. You, J.-Y. Liou, and T.-Y. Ho, "A top-down synthesis methodology for flow-based microfluidic biochips considering valve-switching minimization," in *Proceedings of the 2013 ACM international symposium on International symposium on physical design*. ACM, 2013, pp. 123–129.

[32] T. Tseng, B. Li, T. Ho, and U. Schlichtmann, "Reliability-aware synthesis for flow-based microfluidic biochips by dynamic-device mapping," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 141:1–141:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744899

[33] W. H. Minhass, P. Pop, J. Madsen, and F. S. Blaga, "Architectural synthesis of flow-based microfluidic large-scale integration biochips," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2012, pp. 181–190.

[34] M. C. Eskesen, P. Pop, and S. Potluri, "Architecture synthesis for cost-constrained fault-tolerant flow-based biochips," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, 2016, pp. 618–623.

[35] Y. Su, T. Ho, and D. Lee, "A routability-driven flow routing algorithm for programmable microfluidic devices," in *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*, 2016, pp. 605–610. [Online]. Available: http://dx.doi.org/10.1109/ASPDAC.2016.7428078

[36] K. Hu, T. Ho, and K. Chakrabarty, "Wash optimization and analysis for cross-contamination removal under physical constraints in flow-based microfluidic biochips," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 559–572, 2016. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2015.2488485

[37] D. Mark, S. Haeberle, G. Roth, F. Stetten, and R. Zengerle, "Microfluidic lab-on-a-chip platforms: requirements, characteristics and applications," *Chem. Soc. Rev.*, vol. 39, pp. 1153–1182, 2010.

[38] H. Chou, M. Unger, and S. Quake, "A microfabricated rotary pump," *Biomedical Microdevices*, vol. 3, 2001.

[39] J. P. Urbanski, W. Thies, C. Rhodes, S. Amarasinghe, and T. Thorsen, "Digital microfluidics using soft lithography," *Lab Chip*, vol. 6, pp. 96–104, 2006.

[40] N. Amin, W. Thies, and S. Amarasinghe, "Computer-aided design for microfluidic chips based on multilayer soft lithography," in *Proceedings of the IEEE International Conference on Computer Design*, 2009.

[41] Y. C. Lim, A. Z. Kouzani, and W. Duan, "Lab-on-a-chip: a component view," *Journal of microsystems technology*, vol. 16, no. 12, December 2010.

[42] K. Chakrabarty and J. Zeng, "Design automation for microfluidics-based biochips," *Journal on Emerging Technologies in Computing Systems*, vol. 1, no. 3, pp. 186–223, October 2005.